ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/286147188

# Data Automata in Scala

**Article** · December 2014

DOI: 10.1109/TASE.2014.37

CITATIONS
29

READS
166

**1 author:**

Klaus Havelund
Jet Propulsion Laboratory / California Institute of Technology
**230** PUBLICATIONS **11,355** CITATIONS

SEE PROFILE

All content following this page was uploaded by Klaus Havelund on 09 April 2020.

The user has requested enhancement of the downloaded file.

# Data Automata in Scala

Klaus Havelund
*Jet Propulsion Laboratory*
*California Institute of Technology*
*Pasadena, USA*
*Klaus.Havelund@jpl.nasa.gov*

***Abstract*—The field of runtime verification has during the last decade seen a multitude of systems for monitoring event sequences (traces) emitted by a running system. The objective is to ensure correctness of a system by checking its execution traces against formal specifications representing requirements. A special challenge is data parameterized events, where monitors have to keep track of the combination of control states as well as data constraints, relating events and the data they carry across time points. This poses a challenge wrt. efficiency of monitors, as well as expressiveness of logics. Data automata is a form of automata where states are parameterized with data, supporting monitoring of data parameterized events. We describe the full details of a very simple API in the Scala programming language, an internal DSL (Domain-Specific Language), implementing data automata. The small implementation suggests a design pattern. Data automata allow transition conditions to refer to other states than the source state, and allow target states of transitions to be inlined, offering a temporal logic flavored notation. An embedding of a logic in a high-level language like Scala in addition allows monitors to be programmed using all of Scala's language constructs, offering the full flexibility of a programming language. The framework is demonstrated on an XML processing scenario previously addressed in related work.**

***Keywords*-runtime verification; monitor; parameterized state machines; internal DSL; Scala; XML;**

## I. Introduction

The purpose of formal methods is to assist in the design and development of correct systems, be they software, hardware, or cyber physical systems. Usually a formal method supports analysis of all execution paths of the system, with resulting scalability issues as a consequence. Runtime verification (RV), also referred to as monitoring, however, is focused on just verifying single executions of the system, typically against some formalized specification. Monitoring can occur *online* as the systems executes, or *offline* by analysis of generated log files. It is desirable if an RV specification logic is expressive and the associated monitors are efficient. RV systems are typically complex, with logics of limited expressiveness. Logics are usually variations of state machines, regular expressions, temporal logics, grammars, or rule-based systems. The ideal logic must enable quantification over data in data-parameterized events, must enable past time logic as well as future time logic, and must enable data aggregation and processing (for example counting).

In this paper we illustrate an automaton concept referred to as *data automata*, also referred to as DAUT, for monitoring data-parameterized events. It is implemented as a *shallow internal DSL* (Domain-Specific Language) in the SCALA programming language, meaning that DSL constructs are composed purely of host language constructs, using the interpreter of the host language. This is in contrast to a *deep internal DSL*, where the program exists as data (AST), and where an interpreter is implemented in the host language. In our case, the DSL is essentially an API in SCALA, but SCALA supports the definition of APIs that look and feel like DSLs. The interesting aspect of this solution is its small implementation, which can be characterized as suggesting a *design pattern* for writing monitors / data-parameterized state machines in SCALA. Since it is an extension of SCALA, all of SCALA's programming features can be used for monitoring, which in practice turns out to be useful for even moderately complex monitoring situations, including analysis of log files.

Our data automata are illustrated by examples inspired by the Amazon E-Commerce Service (ECS), which has been discussed and specified in [1] using the domain-specific temporal logic LTL-FO$^+$. Here messages between clients and a server are XML messages, and LTL-FO$^+$ supports formulas over such. We shall illustrate how SCALA's support for XML can be used for obtaining equivalent specifications in DAUT.

The paper is organized as follows. Section II outlines related work. Section III presents the internal DSL through a collection of example properties. Section IV describes the implementation of the DSL. Section V concludes the paper.

## II. Related Work

Data automata were first introduced in [2], where an internal DSL was presented briefly and listed in full in an appendix. The DSL presented in this paper is slightly different, as motivated by the Amazon web-service case study. DAUT is conceptually closely related to the external DSL LOGSCOPE [3], and specifically to the internal SCALA DSL TRACECONTRACT [4]. It is a simplification of TRACECONTRACT by focusing purely on automata (TRACECONTRACT also supports LTL), but goes beyond by adding the

possibility of expressing past time properties in a more uniform manner. Our earlier work includes the rule-based systems LOGFIRE [5], also an internal SCALA DSL, and its predecessor RULER [6]. Rule-based systems appear to be ideal wrt. expressive power (short of a programming language), and are attractive for that reason, but they are also more complex to implement.

Early monitoring systems handling data-parameterized events include [7]–[10]. Of these, MOP [10] is the most efficient, based on *parametric trace slicing*: a trace of data carrying events is sliced to a set of propositional traces. This approach results in an impressive performance, however, at the price of some lack of expressiveness, as pointed out in [11]. MOPBOX [12] is a modular JAVA library for monitoring, implementing MOP's algorithms. ORCHIDS [13], is a comprehensive state machine based monitoring framework created for intrusion detection. Several systems have appeared that monitor first order extensions of propositional linear temporal logic (LTL). These extensions include [14], an embedding of LTL in HASKELL; as well as [1], [15]–[18],

## III. INTRODUCTION TO DATA AUTOMATA

The scenario we shall adopt for illustrating data automata is the Amazon E-Commerce Service, which is described and formalized for runtime verification in [1] using the logic LTL-FO$^+$, an extension of LTL providing first-order quantification over the data inside a trace of XML messages. We have chosen this scenario in order to illustrate the use of SCALA's XML processing capabilities for writing monitors over XML message streams. This is, however, only a secondary point of the presentation. The system in [1] is implemented as a JAVA applet, named BEEPBEEP. BEEPBEEP is demonstrated on the client side, by analyzing messages sent to and received from the server, and by possibly blocking messages or calling user defined functions if violations are found. However, a monitor can be placed on the server side. It is also possible to simply analyze produced logs offline.

The Amazon service makes Amazon.com's inventory available through a web service interface. In addition to simple search and browsing functionalities, ECS also provides shopping cart manipulation operations that allow a client to create a shopping cart, and to add and delete items to and from it. Assume that a cart is identified by a cart Id '*c*', that '*its*' is a list of shopping items, and that '*txt*' is a string. The operations supported by the service include the following (for our example), where arrows indicate direction of messages ($\rightarrow$ from client to Amazon server, and $\leftarrow$ from server to client):

| | | |
|---|---|---|
| *ItemSearch(txt)* | $\rightarrow$ | search items on site |
| *CartCreate(its)* | $\rightarrow$ | create cart with items |
| *CartCreateResponse(c)* | $\leftarrow$ | get cart id back |
| *CartGetResponse(c, its)* | $\leftarrow$ | result of get query |
| *CartAdd(c, its)* | $\rightarrow$ | add items |
| *CartRemove(c, its)* | $\rightarrow$ | remove items |
| *CartClear(c)* | $\rightarrow$ | clear cart |
| *CartDelete(c)* | $\rightarrow$ | delete cart |

Such messages appear as XML messages in Amazon's web-service. For example, a *CartAdd(1, ⟨ 10, 20 ⟩)* message may have the following format[1]:

```
<CartAdd>
  <CartId>1</CartId>
  <Items>
    <Item>
      <ASIN>10</ASIN>
    </Item>
    <Item>
      <ASIN>20</ASIN>
    </Item>
  </Items>
</CartAdd>
```

We shall in the following illustrate two approaches to monitor such XML messages. In the first approach, we design **case** classes (a special form of classes in SCALA) representing these events together with a parsing function, which creates objects of these classes from strings containing the XML messages. In the second approach we will write properties directly over the XML messages. Although the latter solution is interesting due to SCALA's support for XML as a data type, the former solution appears preferable, as shall be discussed.

### A. Events as Case Classes

The event kinds introduced above can be represented in SCALA as **case** classes, as illustrated in Figure 1. Objects of a case class can be created without the use of the **new** keyword, and more importantly: can be used in pattern matching, which turns out to be essential for the elegance of our DSL. Objects of these classes can be generated from strings submitted between server and clients containing XML messages. Figure 2 presents a function *xmlStringToObject*, which transforms a string containing an XML message to an object of one of the classes in Figure 1. The function refers to two auxiliary functions *getId* and *getItems*, which extract respectively the cart id and the shopping items from an XML message using SCALA's implementation of XPATH expressions [19]. The term $x \setminus$ *"str"* extracts from the first inner layer of the XML node $x$ the sequence of nodes of the form `<str> ... </str>`. Furthermore, *x.text* for a given

[1]Amazon's Standard Identification Numbers (ASIN) are here for simplification just small integers.

atomic XML node *x* (having no further nesting) returns the text it contains.

```
case class Item( asin : String )

trait Event
case class ItemSearch( text : String ) extends Event
case class CartCreate ( items : List [Item]) extends Event
case class CartCreateResponse( id : Int ) extends Event
case class CartGetResponse(id: Int , items : List [Item])
  extends Event
case class CartAdd(id: Int , items : List [Item])
  extends Event
case class CartRemove(id: Int , items : List [Item])
  extends Event
case class CartClear ( id : Int ) extends Event
case class CartDelete ( id : Int ) extends Event
```

Figure 1. Case classes representing types of events

```
def xmlToObject(xml:scala.xml.Node):Event =
  xml match {
    ...
    case x @ <CartAdd>{ _* }</CartAdd> ⇒
      CartAdd(getId(x), getItems(x))
    ...
  }

def xmlStringToObject(msg:String ): Event = {
  val xml = scala.xml.XML.loadString(msg)
  xmlToObject(xml)
}

def getId (xml: scala . xml.Node): Int =
  (xml \ "CartId").text.toInt

def getItems (xml: scala . xml.Node): List [Item] =
  (xml \ "Items" \ "Item" \ "ASIN").
    toList . map(i ⇒ Item( i . text ))
```

Figure 2. Transforming XML to objects

We now proceed to formalize the following five properties, the first four of which were also formalized in [1] using LTL-FO$^+$. Property 5 is introduced to illustrate the need for past time logic, which LTL-FO$^+$ does not support.

- **Property 1** - *Until a cart is created, the only operation allowed is ItemSearch.*
- **Property 2** - *A client cannot remove something from a cart that has just been emptied.*
- **Property 3** - *A client cannot add the same item twice to the shopping cart.*
- **Property 4** - *A shopping cart created with an item should contain that item until it is deleted.*
- **Property 5** - *A client cannot add items to a non-existing cart.*

A DAUT monitor defines a set of data parameterized states, and identifies which of these are initial. A state is in part characterized by a transition function representing the transitions leading out of the state. There are various forms of states that can be defined, corresponding to the classical temporal operators known from linear temporal logic [20]. Assume transition functions $ts$, $ts_1$, and $ts_2$, and assume for a given transition function $ts$ that $\lceil ts \rceil$ is the corresponding LTL formula (this is not a formal argument, but serves illustration only). Then DAUT offers the following states (with the corresponding LTL formulas in parenthesis): *always*$\{ts\}$ ($\Box\ \lceil ts \rceil$), *hot*$\{ts\}$ ($\Diamond\ \lceil ts \rceil$, usually referred to as *eventually)*, *next*$\{ts\}$ ($\mathbf{X}\ \lceil ts \rceil$), *wnext*$\{ts\}$ (weak version of **X**), *until*$\{ts_1\}\{ts_2\}$ ($\lceil ts_1 \rceil\ \mathcal{U}\ \lceil ts_2 \rceil$), *unless*$\{ts_1\}\{ts_2\}$ ($\lceil ts_1 \rceil\ \mathcal{W}\ \lceil ts_2 \rceil$), and finally a state *watch*$\{ts\}$ that just waits for one of the transitions in $ts$ to fire, upon which the state is left. Versions of these functions with capital initial letters, for example *Always*, define such states as initial states of a monitor.

Various shorthands allow such monitors to have the flavor of temporal logic specifications, which we shall illustrate first. Properties 1-4 are formalized in Figure 3. Each property is defined as a class, which extends the class *Monitor*, which itself is parameterized with the event type, and which offers a collection of methods for defining properties.

Property 1 is defined as containing one single state, a so-called *unless* state, which is defined by two sets of transitions. The first set of transitions are applied to each incoming event (if they are defined for that event), unless the second set of transitions are able to fire. In this case: unless a cart is created, only *ItemSearch* events are permitted. Transitions are modeled using SCALA's partial functions, which are defined in between curly brackets using pattern matching **case** statements. Unless is a weak until, where the second set of transitions do not have to eventually fire.

Property 2 contains one so-called *always* state, which is always active, and which contains one transition which fires upon observation of a *CartClear* event, binding the parameter to the variable *c*. Upon firing this transition an *unless* state is entered, which is active until a *CartAdd* event occurs with the same cart id *c* (the fact that it must be the same is indicated with quotes around the variable). Any *CartRemove* event with the same cart id triggers an error until then.

Property 3 expresses that upon a *CartCreate(items)* event, then in the *next* state, if a *CartCreateResponse(c)* event is received, providing the identification of the cart created, then from then on any items added with a *CartAdd('c', items_)*

must be disjoint from the originally added items. This is how the property is defined in [1]. The property is, however, conservatively formulated since it does not take into consideration the removal of items.

Property 4 expresses that when items are added to a cart, then for every item *i* added (using SCALA's **for**-**yield** construct) a monitor *unless* state is created, which checks that any response to a get-query asking for the contents of the cart returns a set of items that contains *i*, unless *i* is removed.

Property 5 is a property that requires reference to the past in a manner not supported by LTL-FO$^+$. Note that in the presence of data parameterized events, representing past time logic in terms of future time logic is not possible (a conjecture), as it is in the propositional case. Figure 4 shows how this property can be formalized in DAUT using an explicit state to record whether a cart with a certain cart identifier has been created or not. In the initial state, upon a *CartCreateResponse(c)* event, a *CartCreated(c)* state is created. *CartCreated* states are objects of a case class defined in the monitor, which is parameterized with the cart id. This state itself is defined as a *watch* state, which goes away on a *CartDelete* event for that cart id. In the initial state, if a *CartAdd(c, _)* is observed, and there is no *CartCreated(c)* state active, it is an error. This demonstrates how parameterized states can be referred to in transition conditions, making it possible to express past time properties.

This approach generally allows for definition of data parameterized state machines, including state parameters which are updated as a result of events. For example we could reformulate property 3 to take removal of items into account, thereby allowing items to be re-added to a cart if they have been previously removed. This is shown in Figure 5. Note how SCALA's **val** (constant definition) and **if**-**else** constructs are used, illustrating how programming and logic can be mixed. The '+−' operator has been user-defined to only add elements from the right-hand side argument that do not already occur in the left-hand side argument. Note also how target states of a transition can be composed with '&', forming a set of states.

### B. Events as XML nodes

The properties 1-4 were in [1] formalized directly over XML messages. SCALA supports XML as a data type with values having the format of XML trees, allowing pattern matching and path expressions (as in XPATH) over such. Figure 6 shows how property 4 can be formalized in DAUT using pattern matching and path expressions over XML terms. The other properties have similar but simpler formulations. The type of XML nodes, *scala.xml.Elem*, is imported, and renamed to *Xml*. As can be seen, the formalization is not as succinct as the one using case classes shown in Figure 3. In general, we believe that it is a better approach to transform XML events to objects of case classes, and write properties over these. This also makes the properties less dependent on the format of the XML messages, allowing an XML-to-object transformer, like the one shown in Figure 2, to handle any changes in formats. Furthermore, SCALA's support for pattern matching over

```
class Property1 extends Monitor[Event] {
  Unless {
    case ItemSearch(_) ⇒ ok
    case _ ⇒ error
  } {
    case CartCreate(_) ⇒ ok
  }
}

class Property2 extends Monitor[Event] {
  Always {
    case CartClear(c) ⇒ unless {
      case CartRemove('c', _) ⇒ error
    } {
      case CartAdd('c', _) ⇒ ok
    }
  }
}

class Property3 extends Monitor[Event] {
  Always {
    case CartCreate(items) ⇒ next {
      case CartCreateResponse(c) ⇒ always {
        case CartAdd('c', items_) ⇒
          items disjointWith items_
      }
    }
  }
}

class Property4 extends Monitor[Event] {
  Always {
    case CartAdd(c, items) ⇒
      for (i ∈ items) yield unless {
        case CartGetResponse('c', items_) ⇒
          items_ contains i
      } {
        case CartRemove('c', items_)
          if items_ contains i ⇒ ok
      }
  }
}
```

Figure 3. Properties 1-4 formalized

```
class Property5 extends Monitor[Event] {
  Always {
    case CartCreateResponse(c) ⇒ CartCreated(c)
    case CartAdd(c, _) if !CartCreated(c) ⇒ error
  }

  case class CartCreated(c: Int) extends state {
    Watch {
      case CartDelete(`c`) ⇒ ok
    }
  }
}
```

Figure 4. Property 5 formalized

```
class Property3Liberalized extends Monitor[Event] {
  Always {
    case CartCreate(items) ⇒ next {
      case CartCreateResponse(c) ⇒
        CartCreated(c, items)
    }
  }

  case class CartCreated(id: Int, items: List[Item])
  extends state {
    Watch {
      case CartAdd(`id`, items_) ⇒
        val newCart = CartCreated(id, items +− items_)
        if (items disjointWith items_)
          newCart
        else
          error & newCart
      case CartRemove(`id`, items_) ⇒
        CartCreated(id, items diff items_)
    }
  }
}
```

Figure 5. Property 3 liberalized

XML terms is not optimal. For example, the formalization in Figure 6 uses a combination of pattern matching and path expressions, which seems sub-optimal.

### C. Combining and Applying Monitors

The monitors presented in Sub-section III-A can be combined and applied to analyze a file stored in XML format as shown in Figure 7.

```
import scala.xml.{ Elem ⇒ Xml }

class Property4XML extends Monitor[Xml] {
  Always {
    case add @ <CartAdd>{_*}</CartAdd> ⇒
      val c = getId(add)
      val items = getItems(add)
      for (i ∈ items) yield unless {
        case res @
          <CartGetResponse>{_*}</CartGetResponse>
          if c == getId(res) ⇒
            getItems(res) contains i
      } {
        case rem @
        <CartRemove>{_*}</CartRemove>
        if c == getId(rem) &&
          (getItems(rem) contains i) ⇒ ok
      }
  }
}
```

Figure 6. Property 4 formalized over XML trees

```
class Properties extends Monitor[Event] {
  monitor(
    new Property1(), new Property2(), new Property3(),
    new Property4(), new Property5())
}

object Main {
  def main(args: Array[String]) {
    val m = new Properties
    val file: String = "..."
    val xmlEvents = scala.xml.XML.loadFile(file)

    for (elem ∈ xmlEvents \ "_") {
      m.verify(xmlToObject(elem))
    }
    m.end()
  }
}
```

Figure 7. Combining and applying monitors

## IV. IMPLEMENTATION

This section describes the implementation of the full DAUT DSL. As demonstrated in Figures 3, 4, and 5, a user-defined monitor extends the class *Monitor*, which is parameterized with the event type. The *Monitor* class is

shown below, leaving out its main parts, which will be introduced in the remaining part of this section (all being inserted at the position of the three dots).

```
class Monitor[E <: AnyRef] {
  val monitorName = this.getClass().getSimpleName()
  var monitors : List[Monitor[E]] = List()
  var states : Set[state] = Set()

  def monitor(monitors:Monitor[E]*) {
    this.monitors ++= monitors
  }
  ...
}
```

The event type *E* must be a subtype of type *AnyRef*, which is SCALA's equivalent of JAVA's type *Object*[2]. The variable *monitorName* contains the name of the user-defined monitor, and is useful for printing error messages, identifying the property in case of violations.

The two main variables of a monitor are: *monitors* and *states*. DAUT supports monitors to contain sub-monitors (representing a simple conjunction of these), allowing hierarchical grouping. The sub-monitors of a monitor are stored in the variable *monitors*, and are added via calls of the method *monitor(monitors:Monitor[E]*)*, which takes a variable length number of monitors as arguments. The variable *states* contains the active states of the monitor. The type *state* of states will be defined subsequently. This set contains initially the initial state(s) of the monitor. Next follows a set of auxiliary definitions.

```
  type Transitions = PartialFunction[E, Set[state]]

  def noTransitions : Transitions = {
    case _ if false ⇒ null
  }

  val emptySet : Set[state] = Set()
```

The type *Transitions* represents transitions out of a state. A value of this type is a partial function, which in SCALA can be defined by a sequence of **case** statements enclosed in curly brackets as we saw in Figures 3, 4, and 5. Such a partial function is only defined for the cases provided. Given a partial function *t* (representing transitions), one can test whether it is defined for a certain value *e* (representing an event) with the Boolean valued expression *t.isDefinedAt(e)*. We shall need a default value, *noTransitions*, representing the transition function that is undefined for all events, the empty transition. We shall also for efficiency reasons keep an empty set, *emptySet*, around, since we shall need such frequently during monitoring. Next we define the type of states.

```
class state {
  var transitions : Transitions = noTransitions
  var isFinal : Boolean = true

  def apply(event:E):Set[state] =
    if (transitions.isDefinedAt(event))
      transitions(event) else emptySet

  def Watch(ts: Transitions) {
    transitions = ts
  }

  def Always(ts: Transitions) {
    transitions = ts andThen (_ + this)
  }

  def Hot(ts: Transitions) {
    Watch(ts); isFinal = false
  }

  def Wnext(ts: Transitions) {
    transitions = ts orElse {
      case _ ⇒ ok
    }
  }

  def Next(ts: Transitions) {
    Wnext(ts); isFinal = false
  }

  def Unless(ts1: Transitions)(ts2: Transitions) {
    transitions = ts2 orElse
      (ts1 andThen (_ + this))
  }

  def Until(ts1: Transitions)(ts2: Transitions) {
    Unless(ts1)(ts2); isFinal = false
  }
}
```

A state contains a transition function *transitions*, which initially is empty. A Boolean flag indicates whether a state is final. As a default, all states are final unless otherwise specified. The *apply* method is essential, and takes as argument an event and returns a set of states, namely the target states of transitions. This set is empty in case no transitions fire, and non-empty if any transitions fire.

What follows is a set of definitions of methods, which define this transition function, depending on what kind of

[2]This restriction forbids for example type *Int* as event type. The limitation is for rather non-important technical reasons and could be eliminated.

state it is. The *Wait* method just stores its argument as the transition function, corresponding to waiting for one of the transitions to fire, and then move on. The *Always* method modifies the transition function by adding the current state, represented by **this**, to the set resulting states, modeling the fact that we stay in the source state even if a transition fires. A *Hot* state is like a *Wait* state except that it is not final. A *Wnext* state (weak next) is defined such that if the argument transition function is not defined for the incoming event, a transition is taken anyway, leaving the state. The *Next* state is similar, except that it is not final. Finally, the *Unless* state takes two transition functions as arguments. For a given event it applies the second transition function if defined, and otherwise the first, if defined, with the addition of the source state to the set of target states, modeling that we stay in this state unless the second transition function applies. The *Until* state is similar, except that it is not final. The predefined leaf states are *ok* and *error*. We define an additional function allowing user-defined error messages to be displayed in case of an error.

```
case object ok extends state
case object error extends state

def error(msg:String): state = {
  println("\n*** " + msg + "\n")
  error
}
```

The monitors in Figures 3, 4, and 5 used a technique of inlined states, where the target of a transition could be any of the states we have discussed so far, however, without naming these states, giving a look and feel of temporal logic. The functions that make this possible are the following, which take transition functions as arguments, and return state objects upon which the corresponding transition update methods are called.

```
def watch(ts: Transitions) = new state {Watch(ts)}
def always(ts: Transitions) = new state {Always(ts)}
def hot(ts: Transitions) = new state {Hot(ts)}
def wnext(ts: Transitions) = new state {Wnext(ts)}
def next(ts: Transitions) = new state {Next(ts)}

def unless(ts1: Transitions)(ts2: Transitions) =
  new state { Unless(ts1)(ts2) }

def until(ts1: Transitions)(ts2: Transitions) =
  new state { Until(ts1)(ts2) }
```

The methods that allow us to define states directly at the top level of a monitor are the following, of which we shall only show those used in the examples (the remaining ones follow the exact same pattern).

```
def initial(s: state) { states += s }

def Always(ts: Transitions) { initial(always(ts)) }

def Unless(ts1: Transitions)(ts2: Transitions) {
  initial(unless(ts1)(ts2))
}
```

The method *initial* takes a state as argument and adds it to the set of active states. The method *Always* adds an always state to the set of active states, and similarly for *Unless*.

Recall how it is possible to refer to active parameterized states in conditions. Figure 4 illustrates this, where the expression *CartCreated(c)*, which is fundamentally a constructor call, creating an object of the case class *CartCreated*, is used as a Boolean expression in a condition on a transition. The following definition makes this possible.

```
implicit def stateAsBoolean(s: state): Boolean =
  states contains s
```

The function *stateAsBoolean* is a so-called **implicit** function, which is not explicitly applied anywhere, but which is implicitly applied by the SCALA compiler to any state (in this case) occurring where a Boolean expression is expected, lifting it to a Boolean expression. In this case, a state is lifted to a Boolean predicate returning true iff. the state is contained in the set of active states. The name of an implicit function has no importance.

Note, that in order to use a parameterized state as a predicate in this manner (being true if the state is in the set of active states), one has to provide all the arguments to the state. In cases where not all arguments are known, the function *stateExists* allows alternatively to search for states that satisfy a predicate, returning true if any such exists.

```
def stateExists(p: PartialFunction[state,Boolean]):
Boolean = {
  states exists (p orElse { case _ ⇒ false })
}
```

The monitors shown in our examples rely on a collection of additional implicit functions, shown below. The functions lift various values to sets of states: the results of transition right-hand sides, and hence allow us to use various different forms of right-hand sides.

```
implicit def ss1(u:Unit): Set[state] = Set(ok)

implicit def ss2(b:Boolean):Set[state] =
```

```
  Set( if (b) ok else error )

implicit def ss3(s: state ): Set[ state ] = Set(s)

implicit def ss4(ss: List[ state ]): Set[ state ] =
  ss.toSet

implicit def ss5(s1: state ) = new {
  def &(s2: state ): Set[ state ] = Set(s1, s2)
}

implicit def ss6( set: Set[ state ]) = new {
  def &(s: state ): Set[ state ] = set + s
}
```

The function *ss1* lifts the Unit value to a set of states. This allows us to use a SCALA statement with side effects on the right-hand side of a transition (not shown in examples). The function *ss2* lifts a Boolean value, used for example in monitors *Property3* and *Property4* in Figure 3. The function *ss3* lifts a state, as demonstrated in most of the monitors shown. The function *ss4* lifts a list of states. This is used in monitor *Property4* in Figure 3, where the SCALA construct '**for** (i ∈ items) **yield** ... ' returns a list of states (since *items* is a list). The functions *ss5* and *ss6* lift a state, respectively a set of states, to an object, which defines an '&' method, which as argument takes another state, and forms a set of states. This is used to form sets of multiple target states using infix notation, as illustrated in Figure 5.

Finally, the *Monitor* class provides the methods *verify(event:E)* , which verifies an individual event, and *end()*, which finishes monitoring, for example, useful when analyzing a log file and the end of the log has been reached. The *verify* method operates on two variables:

```
var statesToRemove : Set[ state ] = Set()
var statesToAdd : Set[ state ] = Set()
```

For each new incoming event these are initialized to the empty set, and are then updated to hold the states to remove, respectively add, as a results of matching all the active states of a monitor against an event, caused by transitions firing in these states. These variables represent the *frontier* of states, which are separate from the *current* active states, thereby avoiding non-deterministic evaluation caused by interference between current and next states. The *verify* method is defined as follows.

```
def verify(event:E) {
  for ( sourceState ∈ states ) {
    val targetStates = sourceState(event)
    if (! targetStates.isEmpty) {
      statesToRemove += sourceState
      for ( targetState ∈ targetStates ) {
        targetState match {
          case ‘error‘ ⇒
            println ("\n*** error!\n")
          case ‘ok‘ ⇒
          case _ ⇒ statesToAdd += targetState
        }
      }
    }
  }
  states --= statesToRemove
  states ++= statesToAdd
  statesToRemove = emptySet
  statesToAdd = emptySet
  for (monitor ∈ monitors) {
    monitor.verify(event)
  }
}
```

The method takes as argument an event and iterates through the current active states, and for each applies the state's transition function. If the set of target states is non-empty, the source state is identified to be removed from the set of current states, and the target states are added, operating on the frontier. Error states cause an error message, while ok states cause no action. When all current states have been processed, the current set of states is updated with the temporary frontier, and finally the sub-monitors are evaluated similarly.

The last method is the *end* method, which issues error messages for all non-final states in the current set of states. This method is supposed to be called in situations where monitoring terminates, for example in offline monitoring of a log file. For online monitoring it may never be called.

```
def end() {
  val hotStates = states filter (!_.isFinal)
  if (! hotStates.isEmpty) {
    println(s"hot $monitorName states:")
    hotStates foreach println
  }
  for (monitor ∈ monitors) {
    monitor.end()
  }
}
```

## V. CONCLUSION

We have presented an internal DSL for data automata in SCALA for monitoring event sequences. A version of this DSL has been presented previously in [2], including a more theoretical presentation. However, this paper details the implementation, which has also been modified compared

to [2]. An earlier definition of such an internal DSL was TRACECONTRACT [4]. The DSL was first demonstrated by programming a collection of monitors for processing streams of XML messages, relating to earlier work on this subject. The full implementation of the DSL was then presented. The implementation is remarkably small considering that experiments carried out in [2] demonstrate that monitors are relatively efficient compared to other systems, except for the most efficient such as the MOP system [10]. The implementation can be considered as suggesting a design pattern for writing monitors with data parameterized state machines, which is a contribution on its own. Concerning future work, one can consider adding disjunction of target states. Furthermore, since the DSL is shallow, it is a challenge to optimize it. Deep internal DSL's are easier to optimize since programs (in the DSL) are data. Future work will explore the space between internal and external DSLs.

## ACKNOWLEDGMENT

The work was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The work was furthermore supported by NSF Grant CCF-0926190.

## REFERENCES

[1] S. Hallé and R. Villemaire, "Runtime enforcement of web service message contracts with data," *IEEE Transactions on Services Computing*, vol. 5, no. 2, pp. 192–206, 2012.

[2] K. Havelund, "Monitoring with data automata," in *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Track: Statistical Model Checking, Past Present and Future (organized by Kim Larsen and Axel Legay), Corfu, Greece, October 8-11. Proceedings*, ser. LNCS (volume TBD), T. Margaria and B. Steffen, Eds. Springer, 2014.

[3] H. Barringer, A. Groce, K. Havelund, and M. Smith, "Formal analysis of log files," *J. of Aerospace Computing, Information, and Communication*, vol. 7, no. 11, pp. 365–390, 2010.

[4] H. Barringer and K. Havelund, "TraceContract: A Scala DSL for trace analysis," in *17th International Symposium on Formal Methods (FM'11), Limerick, Ireland, June 20-24, 2011. Proceedings*, ser. LNCS, vol. 6664. Springer, 2011, pp. 57–72.

[5] K. Havelund, "Rule-based runtime verification revisited," *Software Tools for Technology Transfer (STTT)*, April 2014, published online.

[6] H. Barringer, D. E. Rydeheard, and K. Havelund, "Rule systems for run-time monitoring: from Eagle to RuleR," *J. Log. Comput.*, vol. 20, no. 3, pp. 675–706, 2010.

[7] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in *VMCAI*, ser. LNCS, vol. 2937. Springer, 2004, pp. 44–57.

[8] V. Stolz and E. Bodden, "Temporal assertions using AspectJ," in *Proc. of the 5th Int. Workshop on Runtime Verification (RV'05)*, ser. ENTCS, vol. 144(4). Elsevier, 2006, pp. 109–124.

[9] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *OOPSLA'05*. ACM Press, 2005.

[10] P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the MOP runtime verification framework," *Software Tools for Technology Transfer (STTT)*, vol. 14, no. 3, pp. 249–289, 2012.

[11] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard, "Quantified Event Automata - towards expressive and efficient runtime monitors," in *18th International Symposium on Formal Methods (FM'12), Paris, France, August 27-31, 2012. Proceedings*, ser. LNCS, vol. 7436. Springer, 2012.

[12] E. Bodden, "MOPBox: A library approach to runtime verification," in *Runtime Verification - 2nd Int. Conference, RV'11, San Francisco, USA, September 27-30, 2011. Proceedings*, ser. LNCS, vol. 7186. Springer, 2011, pp. 365–369.

[13] J. Goubault-Larrecq and J. Olivain, "A smell of ORCHIDS," in *Proc. of the 8th Int. Workshop on Runtime Verification (RV'08)*, ser. LNCS, vol. 5289. Budapest, Hungary: Springer, 2008, pp. 1–20.

[14] V. Stolz and F. Huch, "Runtime verification of concurrent Haskell programs," in *Proc. of the 4th Int. Workshop on Runtime Verification (RV'04)*, ser. ENTCS, vol. 113. Elsevier, 2005, pp. 201–216.

[15] V. Stolz, "Temporal assertions with parameterized propositions," in *Proc. of the 7th Int. Workshop on Runtime Verification (RV'07)*, ser. LNCS, vol. 4839. Vancouver, Canada: Springer, 2007, pp. 176–187.

[16] D. A. Basin, F. Klaedtke, and S. Müller, "Policy monitoring in first-order temporal logic," in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, Proceedings*, ser. LNCS, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Springer, 2010, pp. 1–18.

[17] A. Bauer, J.-C. Küster, and G. Vegliach, "From propositional to first-order monitoring," in *Runtime Verification - 4th Int. Conference, RV'13, Rennes, France, September 24-27, 2013*, ser. LNCS, vol. 8174. Springer, 2013, pp. 59–75.

[18] N. Decker, M. Leucker, and D. Thoma, "Monitoring modulo theories," in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Grenoble, France, April 7-11, 2014. Proceedings*, ser. LNCS, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 341–356.

[19] "XML XPath 2.0 website. http://www.w3.org/TR/xpath20."

[20] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 1977, pp. 46–57.

View publication stats